4-24-2013

moreBugs: A New Dataset for Benchmarking Algorithms for Information Retrieval from Software Repositories

Shivani Rao
School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN campus, sgrao@purdue.edu

Avinash Kak
School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN campus, kak@purdue.edu

moreBugs: A New Dataset for Benchmarking Algorithms for Information Retrieval from Software Repositories

Shivani Rao

Avinash Kak

TR-ECE-13-07

April 24, 2013

Purdue University

School of Electrical and Computer Engineering

465 Northwestern Avenue

West Lafayette, IN 47907-1285

moreBugs: A New Dataset for Benchmarking Algorithms for Information Retrieval from Software Repositories

Shivani Rao
School of Electrical and
Computer Engineering
Purdue University
West Lafayette, IN
Email: sgrao@purdue.edu

Avinash Kak
School of Electrical and
Computer Engineering
Purdue University
West Lafayette, IN
Email: kak@purdue.edu

***Abstract*—This report presents `moreBugs`, a new publicly available dataset derived from the AspectJ and JodaTime repositories for the benchmarking of algorithms for retrieval from software repositories. As a case in point, `moreBugs` contains all the information required to evaluate a search-based bug localization framework — it includes a set of closed/resolved bugs mined from the bug-tracking system, and, for each bug, its patch-file list and the corresponding snapshot of the repository extracted from version history. `moreBugs` tracks commit-level changes made to a software repository along with its release information. In addition to the benchmarking of bug localization algorithms, the other algorithms whose benchmarking `moreBugs` should prove useful for include: change detection, impact analysis, software evolution, vocabulary evolution, incremental learning, and so on.**

I. Introduction

It goes without saying that the research questions that can be posed on any data-mining related topic are circumscribed by the features incorporated in the datasets that are publicly available to test and validate the hypotheses[1] Recently researchers have used text mining tools to mine information from version control systems like CVS, SVN and Git and bug tracking systems like SourceForge and Bugzilla to create *ground-truth* or *evaluation* or *oracle* datasets for evaluating novel approaches to software defect prediction [3], impact analysis [4], duplicate bug retrieval [5], etc.

With regard to the benchmarking datasets for fault localization algorithms, two such prominent datasets are: (a) SIR (Software-Artifact Infrastructure Repository) and (b) the `iBugs` dataset created by Dallmeier and Zimmerman [6]. SIR contains 130 bugs that are artificially induced. SIR is meant mainly for the evaluation of static and dynamic bug localization techniques. Similarly, the `iBugs` dataset was created to support the evaluation of static and dynamic bug localization techniques and contains real bugs from three software projects: AspectJ, Rhino, and JodaTime. Since the goal of `iBugs` was to support evaluation of static and dynamic bug localization techniques, this dataset only includes a pre-fix and Post-fix snapshot of the repository for each bug and the corresponding test suites.

While intended originally for the evaluation of just static and dynamic bug localization, note, however, that the `iBugs` dataset has been used recently for research in IR based approaches for bug localization [7] [8]. Attempting to use `iBugs` for the evaluation of IR based algorithms has led to a realization of the shortcomings of the dataset for that purpose. One of the main limitations of `iBugs` is that it is based on the version histories at the revision points that only correspond to the big fixes. `iBugs` ignores important information from the version histories such as release history, commit level changes, commit messages, and so on. While the additional information available in version histories may not be useful for static and dynamic bug localization techniques, these can aid in code understanding [1], [2], [9]–[11].

While `iBugs` has served the community well during the last five years, we believe it is time to raise the bar with regard to the "richness" of what is contained in such datasets so that a broader set of research questions can be posed. In addition to bug localization and prediction, we expect this larger set of questions to involve impact analysis [4], software vocabulary (lexicon) evolution [12], and so on. We believe that answering these questions is going to require that a dataset include not only the information that is commonly gleaned from the bug tracking systems, but also the information that is recorded by the versioning tools for *every* revision of a software library. Our new dataset, `moreBugs`, is an answer to these questions.

With regard to the datasets needed for the evaluation of IR-based bug-localization algorithms [13] [14] [15], one needs to mine a set of closed/resolved bugs from the bug-tracking system of a software project and for each bug extract (a) the pre-fix snapshot of the repository from the version history; (b) the textual content of the bug reports to form the queries;

[1]When such benchmark datasets are not available, an alternative technique for demonstrating the effectiveness of new algorithms consists of user studies [1] [2]. However, collecting user-feedback to evaluate a new tool is not always convenient because it requires that a group of users be first trained to use the tool. There are several limitations to the feedback provided by such users: the number of users involved, their prior experience with similar tools, various aspects of their technical and personal background, etc.

TABLE I: moreBugs Specifications

	AspectJ	JodaTime
Version Control System	Git	Git
Number of tags/releases	77	32
Number of revisions	7477	1537
Total duration of the project analyzed	02/12- 12/2	03/12-12/6
Average number of source files/revision	3177	373
Bug tracking system	Bugzilla	SourceForge
Number of bugs from VCS	450	57
Number of bugs in Bugzilla	350	45

and (c) the source files that were changed in order to close the bug. The source files in (c) are needed for the purposes of evaluating the effectiveness of a bug localization algorithm. In the past, researchers have created such benchmark datasets by mining version histories and bug-tracking systems of Eclipse, SWT, ZXing [8], Mozilla Rhino [16], and so on. Unfortunately these datasets are not made publicly available except for a list of bugIDs used to identify the bugs. This makes it impossible to reproduce the results reported.

In this we paper, we present moreBugs, a benchmark dataset that is constructed by mining two different open source projects: AspectJ and JodaTime. As summarized in Table I, moreBugs is based on mining over 10 years of development history for both these projects. We linked the commit messages stored in the respective version tools with the metadata information in the bug tracking system by simple pattern matching techniques. In addition to providing the raw source files, moreBugs also provides the parsed versions that are formatted as TREC style XML files. This is to facilitate experimentation with open-source retrieval tools like Lucene and Lemur. Since mining unstructured version histories from the version tools is time-consuming, cumbersome, and prone to inconsistencies and errors [17], we hope moreBugs will help researchers focus on the implementation of ideas instead of on the parsing of the source files that we had to undertake in order to construct the dataset. Last but not the least we have documented commit-level changes as well as release information. Thus, in addition to bug localization and prediction, we expect this dataset to be useful to researchers that aim to answer research questions involving impact analysis [4], software vocabulary (lexicon) evolution [12], and so on.

II. Research Directions Likely to Benefit from moreBugs

Later in this paper, we will provide a detailed account of all the information that is mined from the version histories and the bug tracking system for a software library. For the purpose of this section, we provide here brief descriptions of the same so that the reader can better appreciate it when we say that the moreBugs dataset would be good for a particular line of research:

- A pre-fix snapshot of the repository for each bug
- The title, description and comments filed in the bug tracking system for each bug
- The set of source files that were fixed in order to resolve the bug (patch-list or change-list)
- The changes made to the software on a revision to revision basis
- Release history of the software
- Commit messages relevant to each commit

With all this information made conveniently available, we believe that moreBugs will be useful for formulating questions and validating hypotheses in at least the following research issues: a) IR based bug localization; b) Vocabulary evolution; and c) Impact analysis.

A. IR Based Bug Localization

IR based bug localization means to locate a bug from its textual description; we want to find the files, methods, classes, etc., that are directly related to the problem causing abnormal execution behavior of the software using retrieval techniques. IR based bug localization approaches are an alternative to the more traditional static and dynamic bug localization techniques [7] [18] [16]. The IR based approaches are not only independent of the programming language and the business concepts of a software system, they are also scalable and extensible to large software systems.

IR based bug localization approaches treat bug localization as a search task, where the bug report is the query and the source files are the database of documents. Thus any benchmark dataset used for an evaluation of an IR based method for bug localization would need a set of bugs and, for each bug, (i) textual content extracted from the bug report; (ii) the set of files fixed in response to that bug report (that would form the "oracle set" for the purpose of evaluation); and (iii) the bug's pre-fix snapshot of the repository which is indexed to create the database of source files. As previously mentioned, evaluation of IR based bug localization techniques has been much facilitated by the availability of the iBugs [6] dataset. Although this dataset was created primarily for the evaluation of static and dynamic bug localization techniques, its usefulness for the evaluation of IR approaches has now been well established [8] [7]. The pre-fix and post-fix snapshot of the software is made available as a part of the repository, but, as we mentioned previously, only at the points of bug fixes.

While the above mentioned list of items extracted for each bug is necessary for any dataset meant for the evaluation of IR algorithms for bug localization, the structured nature of software repositories allows one to incorporate additional information that enables testing of more sophisticated retrieval algorithms — along the lines of recent work in [18], [19], and [20] — that depend on both version histories and bug tracking reports. The moreBugs dataset can be expected to give a boost to this new line of research by a more thorough incorporation of the version histories. This new dataset pulls in the version histories over a time period of 10 years for AspectJ and JodaTime, while recording the changes made at each commit, and linking the bug fixing information with the revision history information.

B. Change or Impact Analysis

Given a bug description or a change request (CR) (or a feature request), change or impact analysis is the process of identifying the software entities that will be potentially impacted as a result of implementing the bug-fix or the CR. Canfora and Cerulo [4] have cast impact analysis as a search task, in that, the proposed CR is treated as a query and the source code entities are treated as documents. The authors have also studied the use of history information available in the commit messages and bug-fixing history in order to improve the model representation of the source code entities for impact analysis. While their study involves three datasets mined from open-source projects, the study suffers from two main shortcomings: (a) The datasets used are not publicly available; and (b) The size of the datasets used ranges from 89 source files to 1538 source files. Also note that the maximum number of CRs used for testing was only 700. On the other hand, `moreBugs` contains 7477 and 1573 CRs in AspectJ and JodaTime respectively. Out of these, 350 of them are linked to bug-fixing in AspectJ and 45 for JodaTime.

C. Vocabulary Evolution

IR approaches to retrieval typically require the creation of a vocabulary set of the terms used in the source repository and the bug reports. Constructing a vocabulary entails stemming, stop-word removal, and handling special situations related to camel-case identifiers (e.g. `PrintToFile`), hard words (e.g `Print_File`), soft words (e.g. `printfile`), the presence of Unicode and other special characters, etc. One may also want to eliminate redundant and ubiquitous terms such as "Copyright" and software constructs that only add noise to the representation of documents and files. Additionally, the terms/words/identifiers of a vocabulary may be classified according to the content in which they appear [12].

Note that the vocabulary associated with a software library is not a static concept since the libraries are typically in a constant state of flux as they are modified in response to bug reports and as new features are added to them. Researchers have studied the impact of software vocabulary evolution on fault proneness [21], software system clustering [22], comments quality, and knowledge discovery and divergence. Software vocabulary evolution has also been studied by Abebe et al. [12]; their case studies were performed by mining version histories of the software over eight releases. Antoniol et al. [23] have studied the evolution of software lexicon and raised important research questions about its relationship to the evolution of program structure for three large software systems over 19 to 24 releases. The `moreBugs` dataset provides the original source files as well as parsed source files for 32 releases for JodaTime and 77 releases for AspectJ. Thus, `moreBugs` should prove useful as an evaluation dataset for vocabulary evolution studies.

III. Related Work

Over the past five to six years, researchers have used text mining tools to create various datasets, often *not* publicly available, for the evaluation of different types of software maintenance tools. Using their own evaluation dataset, Fischer et al. [9] have shown that information available from bug tracking system can be mined and used for enhancing the version information and together they can be used for reasoning about the past and anticipating the future evolution of software projects. They used Mozilla's CVS as the version control system and Bugzilla as the bug tracking system for their study. Canfora and Cerulo [4] have mined version histories from CVS and the information contained in the Bugzilla bug tracking system for three open-source projects — Gedit, ArgoUML, and Firefox — in order to evaluate their impact analysis algorithm. Open source projects like Mozilla, JEdit, Rhino and Eclipse have been mined for evaluation IR-based bug localization techniques using models like, LDA, LSA, Unigram and VSM [13], [16] [7]

There do exist publicly available datasets for the evaluation of tools for fault prediction and traceability link recovery. Traceability link recovery tasks help link the requirements to source files and fault prediction estimates the bugginess of classes and source files in the future releases by studying the patterns in the past and current releases. Zimmermann et al. [3] mined version histories in order to create a dataset for evaluating defect prediction techniques. These datasets however, do not work with version to version changes, but metrics computed on classes from one release to the other.

IV. How `moreBugs` was Created

We now present what went into the creation of `moreBugs`. As previously mentioned, this benchmark dataset was derived from the AspectJ and JodaTime repositories. Both of these projects are now managed by Git, a new "Decentralized Source Code Management" (DSCM) system that has emerged as a popular alternative to CVS and SVN for open-source projects [25]. It is worth noting that there are some major differences between Git, on the one hand, and the more traditional versioning tools like SVN and CVS. Whereas Git allows for a completely decentralized development of code where the information is allowed to flow privately between any two developers (rather than through a centralized system as in SVN and CVS). Since Git is decentralized, it is possible for there to exist several different versions of a software library, all at the same time. When such is the case, one would want to track only the "official repository", the one hosted at the *github*.

In what follows we enumerate each of the steps taken to create the benchmark dataset. The step zero in any analysis and mining of a versioning system is the checking out of the repository from the github and then syncing to the `HEAD`:

- `git clone github`
- `git checkout HEAD`

Figure 1 shows the organization of the data collected in `moreBugs` for a software project.

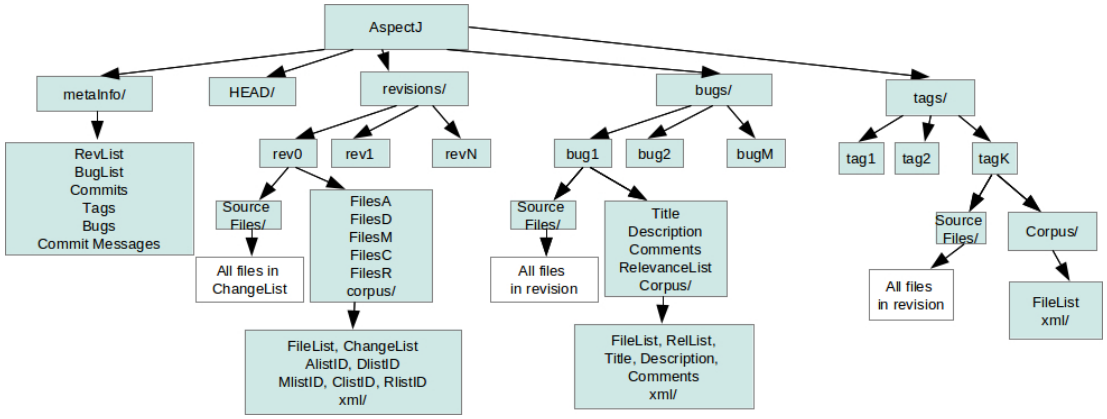

Fig. 1: The organization of the `moreBugs` dataset for each of software projects: AspectJ and JodaTime

A. Log Analysis

The Git log contains valuable information pertaining to the commits made by the developers. The log has all the information related to a commit: the author, the date/time, and the changes made to the software for a particular commit operation (as illustrated in Figures 2 and 3). Each commit is identified by a unique SHA-1 hash code that we denote by *commitID*. The log also encodes the information about the tags and the bugs in the commit messages. For example, in Figure 2, the tag information is shown next to the *commitID*. Similarly, the *bugID* is included in the commit messages by enclosing it in square brackets (Figure 3). Thus by simple pattern matching, it is possible to find out if a commit is related to a bug or a tag. Since Git's commits are identified by a SHA-1 hash value, we assign to it an SVN style revision number that we denote by *revID* for each *commitID*.

Thus by performing basic pattern matching on the log we are able to create the following three tables: `commit_revs`, `tags_revs` and `commit_bugs`. Sample entries from each of these tables are shown in Table II for JodaTime. As this table illustrates, a bug fix may involve multiple commits. Thus it is important to create a fourth table called `prefix_bug` that records the revision number $revID$ of the pre-fix revision for that bug.

In order to track the evolution of a software project, we first find the list of files affected by each commit using the `git diff` command. Source files that are affected by a commit are typically added, modified, deleted, renamed or copied. If *commitID1* and *commitID2* are the consecutive commits and *revID1* and *revID2* the corresponding revision numbers, then `git diff -name-status -C` *commitID1..commitID2* gives us the list of files that are affected at revision number *revID2*. We parse these changes by pattern matching to detect if a file listed in the *change-list* belongs to one of the following categories: *added, modified, deleted, copied, renamed*. The *change-list* as well as the files in each of these categories are documented separately as `FilesA.txt, FilesM.txt, FilesD.txt, FilesC.txt` and `FilesR.txt` (as shown in Figure 1).

TABLE II: Sample data structure entries for JodaTime

commit_revs	
commitID	*revID*
"7394...58a46"	643
"08c3...5a006"	644
commit_bugs	
bugID	*revID*
2827359	1334
2465359	1338
2465359	1339
tags_revs	
revID	*tagName*
185	PRE_PARTIAL_REFACTOR
352	WITH_MILLIS_DURATION
prefix_bug	
bugID	*revID*
2827359	1334
2465359	1338

By using Git's log and diff feature, we have so far explained the data collection process pertaining to extraction of the metadata information.

B. Check out files

In this step, we actually check out the affected files for each revision into the `revisions/revID/sourceFiles` directory. For the revisions that are tagged or labeled as the pre-fix revision for a bug, we check out all the files into a `tags/`*tagName*`/sourceFiles` or `bug/`*bugID*`/sourceFiles` directory. Given the commit hash for a revision, we check out the corresponding snapshot of the repository using the command `git checkout` <*commitID*>. The affected files or the entire repository is then copied to the `sourceFiles/` folder.

C. A Suitable Data Structure to Represent the Changes

In order to track the history of the files we use a table that we refer to as the `fileList` table. Table III shows some sample entries in this data structure. The first field is used to record *fileID*, a number assigned to each file. The second field contains the actual file name. The third field records the last revision where the file was modified. This makes it convenient

```
commit 8fedb261410c9ecfbefeaf864c11de8334bd5bca (tag: v0.9)
Author: Stephen Colebourne <scolebourne@joda.org>
Date:   Tue Dec 16 22:14:09 2003 +0000
    Setup joda-time
    git-svn-id: https://joda-time.svn.sourceforge.net/
    svnroot/joda-time/trunk@5 1e1cfbb7-5c0e-0410-a2f0-f98d92ec03a1

 JodaTime/build.properties.sample |    6 +
 JodaTime/build.xml               |  226 ++++++++++++++++++++++
 JodaTime/checkstyle.xml          |   79 ++++++++++++
 JodaTime/maven.xml               |    4 +
 JodaTime/project.properties      |    8 ++
 JodaTime/project.xml             |  101 +++++++++++++++++
 6 files changed, 424 insertions(+), 0 deletions(-)
```

Fig. 2: An excerpt from the Git log for JodaTime indicating the version number in the commit line at the top

```
commit 66c4a1c10d33243fb0a6f850d350226ab4d55a6c (HEAD, origin/master, origin/HEAD, master)
Author: Stephen Colebourne <scolebourne@joda.org>
Date:   Wed Jun 6 11:35:56 2012 +0100

    Change some StringBuffer uses to StringBuilder [3532330]

 RELEASE-NOTES.txt                                  |    1 +
 .../org/joda/time/IllegalFieldValueException.java  |    2 +-
 src/main/java/org/joda/time/Partial.java           |    2 +-
 .../java/org/joda/time/chrono/BasicChronology.java |    2 +-
 .../java/org/joda/time/convert/ConverterSet.java   |    2 +-
 .../java/org/joda/time/format/DateTimeFormat.java  |    2 +-
 .../java/org/joda/time/tz/ZoneInfoProvider.java    |    2 +-
 7 files changed, 7 insertions(+), 6 deletions(-)
```

Fig. 3: An excerpt from the Git log for JodaTime indicating the *bugID* information embedded in the commit line

```
M     JodaTime/src/java/org/joda/time/chrono/package.html
A     JodaTime/src/java/org/joda/time/convert/AbstractConverter.java
A     JodaTime/src/java/org/joda/time/convert/CalendarConverter.java
C085 JodaTime/src/test/org/joda/test/time/chrono/gj/TestJulianDayOfMonthField.java
                    JodaTime/src/java/org/joda/time/convert/Converter.java
A     JodaTime/src/java/org/joda/time/convert/ConverterManager.java
```

Fig. 4: An excerpt from the change history for revision 9 of JodaTime

to visit the previous revision of a particular source file. The last field indicates if the file is deleted or not. For example, if $D == 1$ for an entry in the table, then the corresponding file does not exist in the repository and the *LastRevID* is the last revision where the file was deleted. The `fileList` table encodes all information that is needed to access files in the current revision of the repository, and helps us detect inconsistencies. For example, if a file is being deleted (as specified by the log in `metaInfo/Git_changeLog/` and for the corresponding entry in the `fileList` table we find that $D = 1$, then it means that a file that was previously deleted is being deleted again, which is an inconsistency. The `fileList` table is updated at each revision based on the changes extracted for that revision.

We should mention that an important side effect of the process that generates this table is the pruning away of certain files whose names do not meet designated extension criteria. We only index and track source files and the related documentation files (readme, xmls, htmls, txts, makefiles) in our `fileList` table. After this round of elimination, some revisions may contain no changes. The revisions that actually do affect files are recorded in the `metaInfo/revList` file.

D. Parse Bug Reports

Next, we turn out attention to bugs. The *bugID*s in the first column of the `bug_prefix` table is used to locate bugs with the same ID as filed in the bug tracking system. Note that not all the bugs mined from the commit logs are found in the bug tracking system. Depending on how sophisticated a bug tracking system is, it may have an XML export facility (as found in Bugzilla, see Figure 5) or one may have to write a parser to scrape the HTML pages that contain information related to the bugs (as for SourceForge). For bugs filed with the SourceForge issue tracking system, we have created a custom

TABLE III: Sample entries of the fileList Structure

fileID	fileName	LastRevID	D
147	"JodaTime/src/test/org/joda/test/time/iso/TestISOMinuteOfDayDateTimeField.java"	3	0
148	"JodaTime/src/test/org/joda/test/time/iso/TestISOMinuteOfHourDateTimeField.java"	3	0
151	"JodaTime/src/test/org/joda/test/time/iso/TestISOSecondOfMinuteDateTimeField.java"	3	0
152	"JodaTime/src/test/org/joda/test/time/iso/TestSuiteISO.java"	3	0
153	"JodaTime/build.xml"	4	0
156	"JodaTime/project.xml"	4	0

parser using Python's `urllib2` and BeautifulSoup libraries to scrape information about a bug given its URL. Typically, any issue/bug tracking system contains information about the time, date of reporting of the bug, the author and the developer it is assigned to, the status of the bug, severity and priority and so on (see Figure 5). Among fields that contain textual information are the title, the description and user comments. We mine the textual information in the title, description and comments for each bug and store it in the folder `bugs/`*bugID*.

E. Creating Relevance List

In order to evaluate bug localization tasks, we need an oracle set of relevant source files that were fixed in order to close a bug. This information may or may not be available depending on the kind of issue tracking system used to track bugs for the software. Bugzilla mandates that the *patch-list* be documented and made available to all users of the software. Sourceforge, on the other hand, makes this information available only to project administrators. In such scenarios, we determine this list from the files that were affected by the commit corresponding to the bug fix. This oracle set *fileID* of the relevant source files for each bug is stored in `Bugs/`*bugID*`/RelevanceList.txt`. Given the `fileList` structure and the names of the source files that belong to the *patchlist* or `RelevanceList`, we create a `RelList` that contains only the $fileID$s of the relevant source files.

F. Cleaning Up of the Raw Source Files For IR Algorithms

So far we have described mining of the Git log, checking out of the relevant source files for each revision bug or a tag, and the creation of an index of the source files via the `fileList` table. We also described parsing of the bug reports and linking them to the version control tool used. We will next discuss one final step: parsing and cleaning up of the source files to make them ready for the modeling and the retrieval steps of IR algorithms.

The parsed source files are stored under their *fileID*s and not *fileNames*. For example, the parsed version of the file `TestISOMinuteOfDayDateTimeField.java` is stored as `corpus-0147.xml` in the `revisions/revID/corpus/xml` and `revisions/revID/corpus/txt` because its *fileID* is 147 according to the `fileList` table (see Table III). The XML files are formatted in TREC-style and can be easily used for experiments with the popular open-source tools used for information retrieval. Each of the source files and documentation files are passed through the following stages of text pre-processing:

- Elimination of Unicode characters and special characters
- Identifier splitting: While splitting camel-case words (`printFile`), hard-words (`print_file`) into individual meaningful terms is trivial, soft-words (e.g.`printfile`) require slightly more involved processing. We have employed greedy methods similar those those developed by Field et al [26] for splitting of soft-words into meaning terms(e.g. `printfile`)

We also need to mention here that we did not perform any sort of stop-word removal or stemming while creating our text-like documents from the source files. This is mainly to aid in flexibility with respect to the two stages of text pre-processing to the users of `moreBugs`.

Parsing source files for bugs and tags: Given the `fileList` table and the other structures that record the evolution of the software, we only need to clean up the affected files at each revision. This step is made efficient as a result of the information stored in the `fileList` table.

V. Datasets Created and Their Stats

As mentioned, we mined two software repositories in order to create `moreBugs`: AspectJ and JodaTime. The specifications of the two datasets thus created are tabulated in Table I. The `moreBugs` dataset is available[2] for download.

In order to get a better understanding of the history of the two projects, we have computed the histogram of the number of source files affected (modified, added, deleted, copied, or renamed) at each revision. We have plotted the histogram of the number of files affected at each revision in Figures 6 and 7 for JodaTime and AspectJ respectively. These figures demonstrate that the frequency of modifications of the source files is higher than the frequencies for the addition or deletion. For a majority of the revisions, at most one file is deleted or added. On the other hand, the distribution of the number of source files modified in a revision is more uniform. The number of revisions when more than five files are affected is very low.

To present some vocabulary-related results obtained with `moreBugs`, Figures 9 and 8 show the histogram of the number of terms affected at each revision for the two projects in the dataset. Note that at most 10 terms are added or deleted a majority of the time. The distribution of the number of terms modified at any revision is slightly more uniform. The findings in this report are very similar to the paper by Haiduc et. al [**?**], in that, the vocabulary changes slowly for a software that has

[2]https://engineering.purdue.edu/RVL/Database/moreBugs/

```
 <bug>
<bug_id>43030</bug_id>
<creation_ts>2003-09-12 12:59:00 -0400</creation_ts>
<short_desc>unit tests failing due to misuse of getAbsolutePath()</short_desc>
<delta_ts>2003-09-17 03:06:00 -0400</delta_ts>
<classification_id>4</classification_id>
<classification>Tools</classification>
<product>AspectJ</product>
<version>1.1.1</version>
<op_sys>Windows XP</op_sys>
<bug_status>RESOLVED</bug_status>
<resolution>FIXED</resolution>
<bug_file_loc/>
<status_whiteboard/>
<keywords/>
<priority>P1</priority>
<bug_severity>normal</bug_severity>
<target_milestone>---</target_milestone>
<everconfirmed>1</everconfirmed>
<reporter name="Jim Hugunin">jim-aj</reporter>
<assigned_to name="Adrian Colyer">adrian.colyer</assigned_to>
<long_desc isprivate="0">
<commentid>185952</commentid>
<who name="Jim Hugunin">jim-aj</who>
<bug_when>2003-09-12 12:59:40 -0400</bug_when>
<thetext>Many of the unit tests in org.aspectj.ajdt.ajc.BuildArgParserTestCase and in
org.aspectj.ajde.StructureModelTest are failing for me.  These failures appear
to be due to the behavior of File.getAbsolutePath().  My suspicion is that these
failures are caused by recent changes to the structure model code to remove
calls to File.getCanonicalPath().
</bug>
```

Fig. 5: Sample bugzilla bug

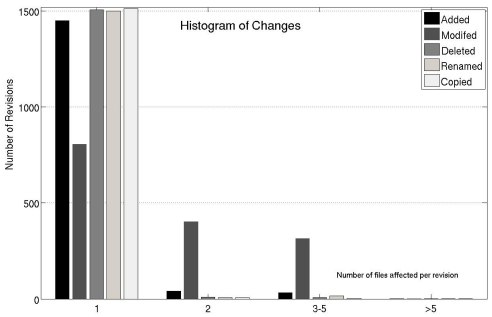

Fig. 6: JodaTime: Histogram of changes on a per-revision basis

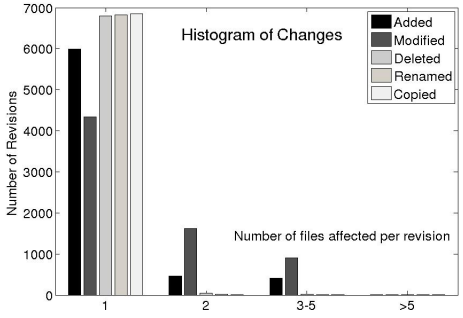

Fig. 7: AspectJ: Histogram of changes on a per-revision basis

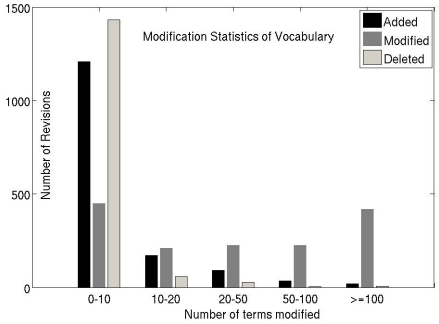

Fig. 8: Histogram of changes made to the vocabulary on a per-revision basis computed using 1573 revisions of JodaTime's history

reached maturation and that new files do not always add new identifier terms to the vocabulary of the software.

VI. How to Use moreBugs

In this section, we illustrate the use of `moreBugs` for the three application areas mentioned in Section II. For two of the three application areas — IR based bug localization and vocabulary evolution — we also present case studies using JodaTime and AspectJ. Table IV shows what part of the information stored in `moreBugs` will be useful for each of these three research areas listed in section II.

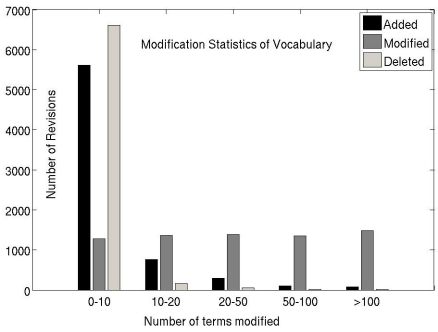

Fig. 9: Histogram of changes made to the vocabulary on a per-revision basis computed using 7477 revisions of AspectJ's history

TABLE IV: Information needs for three applications of moreBugs

	A	B	C
Bug description	Yes	Yes	No
Fixed files	Yes	Yes	No
Commit messages	Yes	No	No
Commit-level Changes	Yes	Yes/No	Yes
Bug-level Changes	Yes	Yes	No
Release Information	No	No	Yes
Parsed Source Files	Yes	Yes	Yes

A. Impact analysis
B. IR based bug localization
C. Software Vocabulary Evolution

A. IR based bug localization

IR based bug localization approaches treat bug localization as a search task, where the bug report is the query and the source files are the database of documents. Thus any benchmark dataset used for an evaluation of an IR based method for bug localization would need a set of bugs and, for each bug, (i) textual content extracted from the bug report; (ii) the set of files fixed in response to that bug report (that would form the "oracle set" for the purpose of evaluation); and (iii) the bug's pre-fix snapshot of the repository which is used to create an index for the purposes of searching.

All the bugs known to `moreBugs` are stored in the directory `Dataset/metaInfo/bugList`. Given the structure of the dataset (Figure 1), the information needed for evaluation of a bug *bugID* is contained in the directory `Dataset/bugs/`*bugID*`/` folder. For each bug in `bugList`, this folder contains the following items:

1) The original source files that are placed in the directory `SourceFiles/` corresponding to the pre-fix snapshot of the software library.
2) The parsed source files in the directories `xml/` and `txt/` are in the TREC-style XML format and plain text format, respectively. The file names are numbered according to the `fileList` table. For example (See Table III), the parsed version of the file `TestISOMinuteOfDayDateTimeField.java` is stored as `corpus-0147.xml`.
3) The three components of a bug description, `Title.txt`, `Description.txt` and `Comments.txt`, are in plain-text format. These serve as the textual content used to query the index of source files.
4) Oracle set of relevant files for evaluation of any bug is listed in the file `RelevanceList`.

As mentioned before, there are three parts of a bug report: title, description and comments, that can be used for retrieval.

B. Vocabulary Evolution

As mentioned earlier, studying vocabulary evolution for a software library gives useful insights into the evolution of the software itself and aids in code understanding. Most of the vocabulary-related experiments have carried out on software libraries have tracked changes from one version (release) to another. In addition, these studies have involved exploring relationships between different kinds of identifiers found in the source code (method names, class names, comments, etc.).

In order to use `moreBugs` for vocabulary evolution, we only need the release history of the software stored in `Database/tags/` folder (See Figure 1). The list of tags is stored in Table `Database/metaInfo/tags_revs`. Each tag points to a snapshot of the software library. The original source files at each release (or tag) can be found in the directory `Database/tags/`*tagName*`/SourceFiles/` and the parsed source files in the directories `Corpus/xml` and `Corpus/txt` folders[3].

C. IR based Impact Analysis

In order to evaluate any novel IR based impact analysis algorithm we need a set of CRs (Change Requests) and for each CRs: (i) description of the CR; (ii) the files that were affected by the CR (including those corresponding to bug(s) related to the CR) to serve as the oracle set as well as to facilitate model-building; (iii) the source files at that revision to serve as the search-able index. All of this information is recorded in `moreBugs` in the `Database/revisions/` folder. The list of revisions can be found in `Database/metaInfo/revList`. For each of the revisions, the set of files affected (added, modified, deleted, renamed or copied) is stored in `revisions/`*revID*`/changeList.txt`. The descriptions of the CRs are extracted from the changelog and stored in `Database/metaInfo/commit_messages.xml` directory in XML format.

VII. Limitations of the current version of the datasets

The primary limitation of our study is that we have only tracked the evolution of certain kinds of files. These are the Java source files and the following documents: HTML, XML, text, README and Makefiles. Files with other extensions are not indexed or tracked. A second limitation arises from the

[3]Again, these source files are named according their placement in the `fileList` table.

fact that we have chosen to eliminate all Unicode and special characters while parsing the source files. In addition, the final vocabulary depends on the choice of the identifier splitting method used. However, this issue can be mitigated by using the raw source files directly and writing a custom parser that suits the need of the task at hand. Third, like any legacy software repository, there is a 15% chance of finding non essential changes in version histories [27]. These changes are redundant mainly because they may involve moving files from one directory to other or renaming files [28] [29] (often called as *rename re-factoring*). While, we have explicitly handled the non-essential changes that occur due to renaming of the files, there could be other non-essential/redundant changes that we did not detect.

A user of `moreBugs` should be aware of the two biases that come into play when mining version histories and bug tracking systems [30]: (1) the commit feature bias, and (2) the bug feature bias. While bug feature bias can be observed or quantified, commit feature bias cannot be measured. It is possible that our datasets suffer from the bug-feature bias.

VIII. Conclusion and Future Work

Considering that `moreBugs` is based on information extracted from 10 years of development for AspectJ and JodaTime, the dataset is relatively large, especially when it is compared to typical benchmarking datasets available today. The AspectJ part of `moreBugs` is derived from 7477 revisions, 77 releases, and 450 bugs, whereas the JodaTime part is derived from 1573 revisions, 32 releases and 57 bugs. We suppose one could say that the large size of `moreBugs` is in consonance with the "big data" focus of modern times.

The main advantage of `moreBugs` is that it provides revision to revision changes that are easy to track using convenient data structures. We believe `moreBugs` will be useful for research in vocabulary evolution, bug localization, impact analysis and so on. The work presented in this paper also illustrates the usability of the dataset for investigating these three research topics.

Note that this is only the first version of `moreBugs`. Our future goal is to create an SQL database instead of mere tables to store and retrieve the metadata information. Note also that so far we have tracked the evolution of the software projects using only their main/trunk branches. We hope to extend our analysis to the other branches of the repositories as well. For the current version, the source files were parsed using simple parsing rules and identifier splitting techniques. For future versions, we plan to experiment with the scr2srcML tool [31]. We also plan to use ChangeDistiller [32] to prune out non-essential changes.

References

[1] A. W. Bradley and G. C. Murphy, "Supporting Software History Exploration," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11, 2011, pp. 193–202.

[2] D. Cubranic, G. Murphy, J. Singer, and K. Booth, "Hipikat: A Project Memory for Software Development," *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 446 – 465, June 2005.

[3] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects for Eclipse," in *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, May 2007, p. 9.

[4] G. Canfora and L. Cerulo, "Fine Grained Indexing of Software Repositories to Support Impact Analysis," in *Proceedings of the 2006 international workshop on Mining software repositories*, ser. MSR '06, 2006, pp. 105–111.

[5] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08, 2008, pp. 461–470. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368151

[6] V. Dallmeier and T. Zimmermann, "Extraction of Bug Localization Benchmarks from History," in *ASE '07: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2007, pp. 433–436.

[7] S. Rao and A. Kak, "Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models," in *Proceeding of the 8th working conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 43–52. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985451

[8] J. Zhou, H. Zhang, and D. Lo, "Where Should the Bugs be Fixed? - More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 14–24.

[9] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, Sept. 2003, pp. 23 – 32.

[10] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06, 2006, pp. 492–501.

[11] A. J. Ko, R. DeLine, and G. Venolia, "Information Needs in Collocated Software Development Teams," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE, 2007, pp. 344–353.

[12] S. Abebe, S. Haiduc, A. Marcus, P. Tonella, and G. Antoniol, "Analyzing the Evolution of the Source Code Vocabulary," in *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, March 2009, pp. 189 –198.

[13] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An Information Retrieval Approach to Concept Location in Source code," in *In Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004*. IEEE Computer Society, 2004, pp. 214–223.

[14] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the Use of Relevance Feedback in ir-based Concept Location," *Software Maintenance, IEEE International Conference on*, vol. 0, pp. 351–360, 2009.

[15] A. Marcus and J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 125–135.

[16] S. K. Lukins, N. A. Karft, and E. H. Letha, "Source Code Retrieval for Bug Localization using Latent Dirichlet Allocation," in *15th Working Conference on Reverse Engineering*, 2008.

[17] D. Port, A. Nikora, J. Hihn, and L. Huang, "Experiences with Text Mining Large Collections of Unstructured Systems Development Artifacts at JPL," in *Software Engineering (ICSE), 2011 33rd International Conference on*, May 2011, pp. 701 –710.

[18] B. Sisman and A. Kak, "Incorporating Version Histories in Information Retrieval Based Bug Localization," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, June 2012, pp. 50 – 59.

[19] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. Nguyen, "A Topic-Based Approach for Narrowing the Search Space of Buggy Files from a Bug Report," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, Nov. 2011, pp. 263 –272.

[20] J. Zhou, H. Zhang, and D. Lo, "Where Should the Bugs be Fixed? More Accurate Information Retrieval-Based Bug Localization Based on

Bug Reports," in *Software Engineering (ICSE), 2012 34th International Conference on*, June 2012, pp. 14 –24.

[21] V. Arnaoudova, L. Eshkevari, R. Oliveto, Y.-G. GueÌĄ andheÌĄ andneuc, and G. Antoniol, "Physical and Conceptual Identifier Dispersion: Measures and Relation to Fault Proneness," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, Sept. 2010, pp. 1 –5.

[22] A. Corazza, S. Martino, V. Maggio, and G. Scanniello, "Investigating the Use of Lexical Information for Software System Clustering," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, March 2011, pp. 35 –44.

[23] G. Antoniol, Y.-G. Gueheneuc, E. Merlo, and P. Tonella, "Mining the Lexicon Used by Programmers during Sofware Evolution," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, Oct. 2007, pp. 14 –23.

[24] J. Cleland-Huang, A. Czauderna, A. Dekhtyar, O. Gotel, J. H. Hayes, E. Keenan, G. Leach, J. Maletic, D. Poshyvanyk, Y. Shin, A. Zisman, G. Antoniol, B. Berenbach, A. Egyed, and P. Maeder, "Grand Challenges, Benchmarks, and Tracelab: Developing Infrastructure for the Software Traceability Research Community," in *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, ser. TEFSE '11, 2011, pp. 17–23.

[25] C. Bird, P. Rigby, E. Barr, D. Hamilton, D. German, and P. Devanbu, "The Promises and Perils of Mining Git," in *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, May 2009, pp. 1 –10.

[26] D. B. H. Field and D. Lawrie., "An Empirical Comparison of Techniques for Extracting Concept Abbreviations from Identifiers." in *Proceedings of IASTED International Conference on Software Engineering and Applications*, 2006.

[27] D. Kawrykow and M. P. Robillard, "Non-essential Changes in Version Histories," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 351–360. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985842

[28] Y. Yu, T. T. Tun, and B. Nuseibeh, "Specifying and Detecting Meaningful Changes in Programs," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, Nov. 2011, pp. 273 –282.

[29] S. Thangthumachit, S. Hayashi, and M. Saeki, "Understanding Source Code Differences by Separating Refactoring Effects," in *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, Dec. 2011, pp. 339 –347.

[30] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and Balanced?: Bias in Bug-fix Datasets," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595716

[31] J. Maletic, M. Collard, and A. Marcus, "Source Code Files as Structured Documents," in *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, 2002, pp. 289 – 292.

[32] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *Software Engineering, IEEE Transactions on*, vol. 33, no. 11, pp. 725 –743, Nov. 2007.